The COMEL Language

Rosziati Ibrahim **Clemens Szyperski**

School of Computing Science
Queensland University of Technology
Brisbane, Australia

ibrahim@fit.qut.edu.au c.szyperski@qut.edu.au

Abstract

In this paper, we propose an exemplary language, COMEL, that support the Microsoft's Component Object Model (COM) facilities. Microsoft's COM is a leading example of the currently emerging technologies for component software. As such, it is the foundation of Microsoft's OLE (Object Linking and Embedding) and ActiveX. Despite its importance, COM does not have a formal specification. An exemplary language, COMEL, is used to model COM's rules. A formal type system for COMEL is presented. This paper in effect presents an approach to the formalization of COM.

Keywords: COM, OLE, COMEL, component software, objects, interfaces, containment, aggregation, type rules, type systems

1. Introduction

Microsoft's OLE (Object Linking and Embedding) provides an application integration framework for Microsoft Windows. OLE rests on the Component Object Model (COM), which specifies a programming language independent binary standard for object invocations, plus a number of interfaces for foundational services.

In Microsoft's OLE, Component Object Model (COM) is used in order to build software. COM is the base upon which OLE is built and the foundation for future Windows-based operating systems, which allows large software applications to be broken into more manageable and maintainable components [Broc95]. In addition, with the arrival of Distributed COM (DCOM), COM is also the foundation for Microsoft's distributed computing approach.

COM is largely a set of rules for building interoperable software components. COM places the burden of support for the protocol onto the developer who must duplicate this support in every COM object he or she implements (for example, the IUnknown interface must exist in every COM object). Since support at runtime is lacking, implementing COM objects is a highly complex, time consuming task with opportunities for misinterpretation and error. Because COM does not support many commonly accepted object-oriented concepts and mechanisms, developers are forced to adopt an unfamiliar programming paradigm and, in fact, typically have to switch between the paradigm provided by the implementation language (for example C++) and the paradigm enforced by COM.

In order to precisely specify COM's rules, which are currently captured by conventions only, COM needs to be formalized. In this paper, a general framework of formalizing type systems will be used as a basis to formalize COM. The formalization of type systems requires the development of precise notations and definitions, and the detailed proof of formal properties that gives confidence in the appropriateness of the definitions [Card96]. Therefore, the first stage in the formalization of COM is to develop a small language with COM-like facilities. Then the type rules are being introduced to this new language. Based on this typed language, a more formal model for COM will be developed. Since COM itself is language-independent, the language introduced here just takes an exemplary role. This new language for COM is called COMEL (Component Object Model Extended Language, pronounced as cho-mell).

The design of the language is quite minimalistic focusing on three main purposes. The first purpose of the language is to totally hide the IUnknown interface in every object. This in consequence will hide reference counting and dynamic interface acquisition. The second purpose of the language is to model the implementation of objects in a COM environment that use containment or aggregation. The final purpose of the language is to try to capture all the necessary detail at the declaration level of the language.

In Section 2, some important facts and concepts of COM are outlined. In Section 3, a context-free grammar for a COM-supporting language is proposed. The informal semantics of the language are described in Section 4. Finally, the type rules for the COMEL language are covered in Section 5.

2. Component Object Model (COM) - A Review

COM allows applications to be turned into OLE-enabled components, and creates binary run-time interfaces between applications in order to enable two different applications to interact with each other. A component corresponds to a module containing multiple classes. A COM object is an instance of such a class. COM objects have state and an identity. All interaction with a COM object is through COM interfaces. A COM class can support any number of interfaces. A COM interface has any number of methods, but no fields. The usual notion of subtyping is replaced by that of supersets of interfaces and by the separation of implementations (classes) from interfaces. Implementation inheritance across components is not supported in COM.

All COM interfaces are subtypes of the IUnknown interface. Interface IUnknown is considered as a base interface in every COM object. The IUnknown interface itself provides QueryInterface, AddRef and Release functions.

The QueryInterface function is used to query an object for specific interfaces. The way it works is as follows. A client must first ask the object whether it supports the interface by calling QueryInterface and providing the GUID (Globally Unique Identifier) of the required interface as a parameter argument to QueryInterface. QueryInterface will then return a pointer into the object instance that implements the indicated interface. If however, the receiver does not contain an implementation of the interface, QueryInterface will return a NULL pointer. The QueryInterface function helps overcoming the problem of versioning by requiring each interface to have an exact and unique GUID. This implies that the interface is immutable - once made public, it can never be changed. Therefore, if an implementor wants to add new functionality to the interface, he or she is required to define a new interface and a new GUID will be assigned to this new interface. This GUID is well known as IID (Interface Identifier). Another important GUID that is used in COM is CLSID (Class Identifier). CLSID is used for the class of a component. In order to create an instance (an object) of the component, a client needs to provide the CLSID of the required component as a parameter argument to CoCreateInstance. CoCreateInctance is a COM library function to create an object.

AddRef and Release functions are used for reference counting. COM does not have proper garbage collection. In order to collect memory back, proper use of the AddRef and Release functions is required. Therefore, it is a responsibility of a client or programmer to call the AddRef function whenever a new interface pointer is created and to call the Release function whenever the interface pointer is destructed. Upon calling, AddRef will increase a reference counter and Release will decrease a reference counter. If a COM object detects that its reference count is 0, it will delete itself. This is done of course after a decrement of reference counting not at start-up of application.

COM is designed around the use of interfaces because interfaces enable interoperability. Interoperability means that the interfaces that are used by objects are strongly typed semantic contracts between client and object. This means that if a client uses an object's interface, client and server "sign" a contract that the interfaced functionality will not change behaviour unexpectedly.

The major difference between COM and other standard object models is that COM does not support implementation inheritance for reuse of existing objects across components. COM classes cannot inherit implementation from each other. Reuse is therefore solely based on object composition rather than class composition. The two forms of object composition supported by COM are called containment and aggregation. Containment refers to the forwarding of requests to another object. Aggregation improves performance in cases where containment would lead to a deep forwarding chain. Instead of forwarding requests, references to another object's interfaces are directly handed out to clients in a way that does not reveal the forwardees identity to the client.

The two special properties of COM, no implementation inheritance and immutable interfaces, set COM off from other object models and are the major motivation for formalizing COM.

2. The COMEL's Syntax

This section proposes the new model language for COM, named COMEL. The COMEL language design follows the model of BOPL by Palsberg and Schwartzbach [Pals94]. The definitions of some of the notations used in the COMEL syntax are as follows:

ide ::= A	the non-terminal identifier ide is defined to be A
ide	a non-terminal symbol
"ide"	a terminal symbol
A B	means A followed by B (binds strongest)
A | B	means A or B
[A]	means 0 or 1 occurrences of A
{A}	means 0 or 1 or more (>=0) occurences of A

The syntax of COMEL is shown in Figure 3-1.

```
COMEL        ::=  CLSIDDecl  IIDDecl  "COMPONENT" Id "=" ClsId
                  "IS"  Contains  Aggregates  Const  Var
                       Methods  Interfaces   "END" Id "."

CLSIDDecl    ::=  [ "CLSID" { Id "=" ClsId ";" } ]

IIDDecl      ::=  [ "IID" { Id "=" IId ";" } ]

Contains     ::=  { "CONTAINS" Id ":" InterfaceType "=" "NEW" Id ";"}

Aggregates   ::=  { "AGGREGATES"  Id "." Id { ","  Id "." Id } ";"  }

Interfaces   ::=  { "INTERFACE" Id  "IS"   Methods  "END" Id ";"  }

Methods      ::=  {   "METHOD" Id Formals [ ":" Type ] "IS"
                      Var  "BEGIN"  StatSeq  "END"  Id ";"  }

Const        ::=  [   "CONST" {  Id "=" Expr ";"  } ]

Var          ::=  [   "VAR" { IdList  ":" Type  ";"  } ]

Formals      ::=  "("  [ IdList ":" Type { ";"  IdList ":" Type }] ")"

StatSeq      ::=  Stat  { ";" Stat }

Stat         ::=  [  "IF" Expr "THEN" StatSeq ["ELSE" StatSeq] END"
                  |  "WHILE" Expr "DO"  StatSeq "END"
                  |  "CASE" Id  "IS"  Alternative { "|" Alternative }
                       [ "ELSE" StatSeq ]  "END"
                  |  "RETURN" [ Expr ]
                  |   Designator [ ":=" Expr | ArgList ]    ]

Alternative  ::=  InterfaceType ":" StatSeq

Expr         ::=  SimpleExpr [ Relation SimpleExpr ]

SimpleExpr   ::=  Term { AddOp Term }

Term         ::=  Factor { MulOp Factor }
```

Factor	::=	Int | Designator ArgList | "(" Expr ")" | "NEW" Id | "NIL" | "FALSE" | "TRUE"
Designator	::=	Id ["." Id] ["→" Id]
ArgList	::=	"(" [ExprList] ")"
ExprList	::=	Expr { "," Expr }
Type	::=	BasicType | InterfaceType
InterfaceType	::=	"{" [IdList] "}"
BasicType	::=	"INTEGER" | "BOOLEAN"
IdList	::=	Id { "," Id }
Relation	::=	"=" | "#" | "<" | ">" | "<=" | ">="
AddOp	::=	"+" | "-" | "OR"
MulOp	::=	"*" | "DIV" | "AND" | "MOD"
ClsId	::=	String
IId	::=	String
String	::=	' " ' {Char} ' " '
Int	::=	Digit { Digit }
Id	::=	Letter { Letter | Digit }
Char	::=	Letter | Digit | "-" | "." | "{" | "}"
Letter	::=	"a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
Digit	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Figure 3-1 : COMEL's Syntax

4. The Informal Semantics

This section gives the informal semantics of the COMEL language. The words in *italics* represent reserved words in the COMEL language. When necessary, an example is presented to help the proper explanation of the meaning of the language.

4.1. COMEL

COMEL ::= CLSIDDecl IIDDecl *COMPONENT* Id "=" ClsId *IS*
Contains Aggregates Const Var
Methods Interfaces
END Id "."

This top-level construct creates a new software component. The name **Id** of the component is bound to a CLSID (Class Identifier). If a component needs to use other components, it needs to declare the CLSID of each component used. In each component the IID (Interface Identifier) of each interface used must be declared. Components can also use the containment or/and aggregation methods. The ConstDecl is meant to be used for global constant declarations. The VarDecl is meant to be used for global variable declarations. Each component can contain none, one or more methods that are local and not meant to be exported out of the component. Each component can also provide none, one or more interfaces that are meant to be exported out of the component.

4.2. CLSID Declaration

CLSIDDecl ::= [*CLSID* { Id "=" ClsId ";" }]

The CLSIDDecl clause is used to declare the CLSID of each component used. In COMEL, everytime a component wants to use another component, the CLSID of the required component must be provided here.

4.3. IID Declaration

IIDDecl ::= [*IID* { Id "=" IId ";" }]

The IIDDecl clause is used to declare the IID of each interface. In COMEL, everytime a component wants to use another interface from another component, the IID of the required interface must be provided here.

4.4. Containment

Contains ::= *CONTAINS* Id ":" InterfaceType "=" *NEW* Id ";"

The Contains clause is used when the component contains other components. This means that the component uses the containment method. Therefore it needs to create an instance of the other component in order to use that component's interfaces. Consider the following example using the Contains syntax.

CONTAINS a : {b,c} = *NEW* d;

The above syntax literally means that a new instance of component **d** is created and bound to a variable **a**. At the same time, it is checked whether component **d** support interfaces **b** and **c**. The CLSID of component **d** needs to be declared using a CLSIDDecl clause.

4.5. Aggregation

Aggregates ::= *AGGREGATES* Id "." Id { "," Id "." Id } ";"

The Aggregates clause is used when the component uses other interfaces that belong to other components as if they were its own interfaces. This means that the component uses the aggregation method. Therefore it needs to expose or aggregate the other component's interfaces. Consider the following example using the Aggregates syntax.

AGGREGATES a.b, a.c;

The above syntax literally means that the interfaces **b** and **c** are directly used (without modification) from component **a**. The Aggregates clause needs to go together with the Contains clause, because aggregation uses an inner object (that is, a component instance) which needs to be first introduced using the Contains clause.

4.6. Interface Declaration

InterfaceDecl ::= { *INTERFACE* Id *IS* MethodDecl *END* Id ";" }

The InterfaceDecl is used to declare the interface(s) that will be used in the component. Each interface can contain none, one or more methods.

4.7. Method Declaration

MethodDecl ::= { *METHOD* Id Formals [":" Type] *IS*
VarDecl *BEGIN* StatSeq *END* Id ";" }

The MethodDecl is used to declare the method(s) that will be used for the interface or component. In each method there can be variable declarations that are local to the method. Each method can contain none, one or more statements that will perform the method task.

4.8. Constant Declaration

ConstDecl ::= [*CONST* { Id "=" Expr ";" }]

The ConstDecl clause is used to declare the constant variables used either in components or in methods. For example, consider the following example using the ConstDecl syntax:

CONST a = 1;

The above syntax literally means that a variable **a** is assigned to 1 and automatically bound to type INTEGER.

4.9. Variable Declaration

VarDecl ::= [*VAR* { IdList ":" Type ";" }]

The VarDecl clause is used to declare the variables used either in components or in methods and is also used to introduce type declarations. The type can either be *BOOLEAN*, *INTEGER*, an Identifier, or a set of Identifiers. For example, consider the following example using the Decl syntax:

VAR a : {b,c} ;

The above syntax literally means that a variable **a** is bound to type "**b** and **c"** where **b** and **c** are interfaces. In COMEL, it allows to assign a variable (an object) that provides interfaces as above syntax - this is just a rule to say that an object **a** provides interfaces **b** and **c**. In proper meaning, this implies that a type of object **a** is a set of interfaces **b** and **c**.

4.10. Formals Argument

Formals ::= "(" [IdList ":" Type { ";" IdList ":" Type }] ")"

The Formals clause is used to declare the formal parameters of a method.

4.11. Statement Sequence

StatSeq ::= Stat { ";" Stat }

The StatSeq represents the statement sequence that can be included in private component methods or in interface methods. The statement sequence can consist of none, one or more statements (Stat can be the empty statement).

4.12. Statement

The following is the informal semantics for each different kind of statement. The COMEL language supports conditionals, iteration, and case statements.

- "*IF* Expr_1 *THEN* $\mathrm{StatSeq}_1$ [*ELSE* $\mathrm{StatSeq}_2$] *END*" first evaluates Expr_1, which must yield a boolean value. If this value is true, then $\mathrm{StatSeq}_1$ is evaluated; otherwise, if the *ELSE* option is used, $\mathrm{StatSeq}_2$ is evaluated; otherwise nothing is evaluated.

- "*WHILE* Expr *DO* StatSeq *END*" first evaluates Expr, which must yield a boolean value. If this value is true, then StatSeq is evaluated and the *WHILE*-loop is iterated; otherwise nothing is evaluated.

- "*CASE* Id *IS* Alternative { | Alternative } [*ELSE* $\mathrm{StatSeq}_2$] *END*", where "Alternative" is "InterfaceType : $\mathrm{StatSeq}_1$", first takes Id and evaluates it to type value (here Id is only permitted to be an instance of object). If this value is of type InterfaceType or a subtype thereof, then $\mathrm{StatSeq}_1$ is evaluated. If other Alternative options are used, it will yield to only one type value of InterfaceType and evaluate only the $\mathrm{StatSeq}_1$ that belongs to the value of that InterfaceType (evaluation is in-order of the first one match); otherwise if ELSE option is used, $\mathrm{StatSeq}_2$ is evaluated.

- "*RETURN* [Expr]" will return value of Expr. The expression should be of the same type as the method, or absent if the method has no type.

- "Designator [":=" Expr | ArgList]" will evaluate to different meaning according to its usage. If the "Designator ":=" Expr" option is used, it will evaluate to an assignment statement of either type Integer or Boolean or Identifier or set of Identifiers . If the "Designator [ArgList]" option is used, it will evaluate to a method call with ArgList as actual parameters. The method may belong to either another interface or another component or the same component, private or same interface. A method call in this way should not return any type.

4.13. TypeCase

Alternative ::= InterfaceType ":" StatSeq

The Alternative syntax is used for the purpose of typecase. Hiding the IUnknown interface in COMEL means that the QueryInterface function cannot be used to query an object for specific interface. This is replaced by a typecase statement for dynamic interface acquisition. For example, consider the following example using the Alternative syntax together with the Case syntax:

CASE obj *IS* {b,c} : *RETURN TRUE*
ELSE RETURN FALSE END;

The above syntax literally means that the object **obj** is checked to see whether it provides interfaces **b** and **c**. If it does, the boolean value *TRUE* is returned; otherwise the boolean value *FALSE* is returned.

4.14. Expression

We begin our informal semantics for expressions by an "induction". For simplicity the following abbreviation will be used:

Int ≡ *INTEGER* and Bool ≡ *BOOLEAN*
Id ≡ Identifier and Set-of-Id ≡ Set of Identifiers
LHS ≡ Left Hand Side and RHS ≡ Right Hand Side

An expression will evaluate to one of the following:

- Expr ::= SimpleExpr { Relation SimpleExpr }
  This syntax can only take one of the following final expression:
  1. x = y where x and y are of type Int or Bool or Id or Set-of-Id and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is equal to RHS y; otherwise the resulting value is *FALSE*.
  2. x # y where x and y are of type Int or Bool or Id or Set-of-Id and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is not equal to RHS y; otherwise the resulting value is *FALSE*.
  3. x < y where x and y are of type Int and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is less than RHS y; otherwise the resulting value is *FALSE*.
  4. x > y where x and y are of type Int and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is greater than RHS y; otherwise the resulting value is *FALSE*.
  5. x <= y where x and y are of type Int and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is less and equal than RHS y; otherwise the resulting value is *FALSE*.

6. x >= y where x and y are of type Int and the resulting value is of type Bool. The resulting value is *TRUE* if LHS x is greater and equal than RHS y; otherwise the resulting value is *FALSE*.
7. If Relation is not in used, the Expr syntax will be the same as SimpleExpr syntax.

- SimpleExpr ::= Term { AddOp Term }

  This syntax can only take one of the following final expression:
  1. x + y where x and y are of type Int and the resulting value is of type Int.
  2. x - y where x and y are of type Int and the resulting value is of type Int.
  3. x *OR* y where x and y are of type Bool and the resulting value is of type Bool. The usual operation of *OR* operator is used.
  4. If AddOp is not in used, the SimpleExpr syntax will be the same as Term syntax.

- Term ::= Factor { MulOp Factor }

  This expression can only take one of the following final expression:
  1. x * y where x and y are of type Int and the resulting value is of type Int.
  2. x *DIV* y where x and y are of type Int and the resulting value is of type Int (rounded Integer).
  3. x *MOD* y where x and y are of type Int and the resulting value is of type Int.
  4. x *AND* y where x and y are of type Bool and the resulting value is of type Bool. The usual operation of *AND* operator is used.
  5. If MulOp is not in used, the Term syntax will be the same as Factor syntax.

- Factor ::= Int | Designator ArgList | "(" Expr ")" | *NEW* Id | *NIL* |
  FALSE | *TRUE*

  1. The trivial cases for this syntax is when it will evaluate to either Boolean expression (*TRUE* or *FALSE*) or Integer value or *NIL* value.
  2. The other expression that this syntax will evaluate at is the *NEW* Id expression. "*NEW* Id" looks up the component named Id in the component library and allocates a new instance (an object) of that component.
  3. If " "(" Expr ")" " option is used, it will evaluate to Expr where this expression will then evaluate to Boolean or Integer or Id or Set-of-Id expression of either *TRUE* or *FALSE*.
  4. If "Designator ArgList" option is used, it will evaluate to a call to a method with ArgList as actual parameters. The method may belong to either another interface or another component or the same component, private or same interface. A method call in this way should return a type of BasicType.

The remaining syntax is self explanatory. A type can be either *INTEGER*, *BOOLEAN* Id, or a set of Ids (interfaces).

5. The Type Rules

This section covers the type rules of COMEL. The basic types of COMEL are as follows:

InterfaceType = $\mathbf{P}$(IID)
Types = {INTEGER, BOOLEAN} $\cup$ InterfaceType $\cup$ MethodType $\cup$ Interface

The following diagram summarizes the terminology used, following Cardelli [Card96] and Plotkin [Plot81]. It also summarizes various auxiliary definitions used to keep the judgments concise. Identifiers in COMEL refer to methods, interfaces, components and variables. Therefore, symbols *M, I, O* and *x* are used to represent Method, Interface, Component and Variable identifiers respectively. Symbol T represents Type and symbol A represents InterfaceType.

$\Gamma \vdash \Diamond$	Γ is a well-formed environment, where an environment is a mapping from identifiers to types
$\Gamma \vdash T$	T is a well-formed type in Γ
$\Gamma \vdash O$	O is a well-formed component in Γ
$\Gamma \vdash D \therefore S$	D is a well-formed declaration of signature S in Γ, where a signature is a mapping from identifiers to types
$\Gamma \vdash Impl : T$	Impl is a well-formed implementation of type T in Γ
$\Gamma \vdash C$	C is well-formed command in Γ
$\Gamma \vdash E : T$	E is a well-formed expression of type T in Γ
$Id \in dom(\Gamma)$	Id is defined in Γ
$O \in dom(\Gamma_{CLSID})$ Γ_{CLSID} : CLSID $\rightarrow$ { }	if O is defined in Γ_{CLSID}, where Γ_{CLSID} is maintained by COM's CLSID repository, then CLSID maps to InterfaceType (All components are statically modelled to have type { }.)
$I \in dom(\Gamma_{IID})$ Γ_{IID} : IID $\rightarrow \Gamma$	if I is defined in Γ_{IID}, where Γ_{IID} is maintained by COM's IID repository, then IID maps to an environment (A mapping from identifiers to method types.)
x.I I$\rightarrow$ M	select interface I of object referred to by x (x $\neq$ NIL) select method M of interface I (**.** binds stronger than $\rightarrow$)

Note that $dom(\Gamma) \cap dom(\Gamma_{IID}) = \phi$, $dom(\Gamma) \cap dom(\Gamma_{CLSID}) = \phi$ and $dom(\Gamma_{IID}) \cap dom(\Gamma_{CLSID}) = \phi$. Type rules assert the validity of certain judgments on the basis of other judgments that are already known to be valid. The general form of a type rule is as follows [Card96]:

(Rule name) $\dfrac{\Gamma_1 \vdash \Im_1 \ \ldots \ \Gamma_n \vdash \Im_n \ \text{[Annotations]}}{\Gamma \vdash \Im}$	Each type rule is written as a number of *premise* judgments $\Gamma_i \vdash \Im_i$ above a horizontal line, with a single *conclusion* judgment $\Gamma \vdash \Im$ below the line. When all the premises are satisfied, the conclusion must hold.

Figure 5-1 summarizes the abstract subset of COMEL's syntax used by the type rules. The words in *italics* represent reserved words in the COMEL language. Then, the type rules for this COMEL's syntax are presented.

Component	::=	*COMPONENT* O *IS* $State_1$ .. $State_n$ $ExposedInterface_1$ .. $ExposedInterface_m$
State	::=	Contains | {Const Var}
Contains	::=	*CONTAINS* x : {I_1 .. I_n} = *NEW* O;
ExposedInterface	::=	Aggregates | Interface
Aggregates	::=	*AGGREGATES* x.I ;
Interface	::=	*INTERFACE* I *IS* {$Method_1$ .. $Method_n$}
Method	::=	*METHOD* M (x_1: T_1 .. x_n: T_n) : T *IS* body;
Const	::=	*CONST* x = Expr;
Var	::=	*VAR* x : Type;
Type	::=	*INTEGER* | *BOOLEAN* | InterfaceType
InterfaceType	::=	{I_1 .. I_n}
body	::=	{Var_1 .. Var_m} Cmd
Cmd	::=	C_1 ; C_2 | x := Expr | *RETURN* Expr | *IF* Expr *THEN* C_1 *ELSE* C_2 *END* | *WHILE* Expr *DO* Cmd *END* | *CASE* x IS {I_1 .. I_n} : C_1 *ELSE* C_2 *END*
Expr	::=	E_1 + E_2 | E_1 - E_2 | E_1 *OR* E_2 | E_1 = E_2 | E_1 # E_2 | E_1 < E_2 | E_1 > E_2 | E_1 <= E_2 | E_1 >= E_2 | E_1 * E_2 | E_1 *DIV* E_2 | E_1 *AND* E_2 | E_1 *MOD* E_2 | *NEW* O | x.I→M(E_1 .. E_n) | Number | *NIL* | *FALSE* | *TRUE*

Figure 5-1 : The abstract subset of COMEL's syntax used by the type rules

The type rules for the syntax from Figure 5-1 are grouped by judgment kinds. Thus:

Environments

(Env ϕ)

$$\frac{}{\phi \vdash \Diamond}$$

(Env Id)

$$\frac{\Gamma \vdash T \quad [Id \notin dom(\Gamma)]}{\Gamma,\ Id : T \vdash \Diamond}$$

(Env Decl)

$$\frac{\Gamma \vdash D \therefore S}{\Gamma,\ S \vdash \Diamond}$$

Types

(Type INTEGER)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathit{INTEGER}}$$

(Type BOOLEAN)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathit{BOOLEAN}}$$

(Type InterfaceType)

$$\frac{[I_j \in dom(\Gamma_{IID}) \quad (j = 1 .. n)]}{\Gamma \vdash \{I_1 .. I_n\}}$$

Subsumption

(COM Subsumption)

$$\frac{\Gamma \vdash a : A_0 \quad [A_1 \subseteq A_0]}{\Gamma \vdash a : A_1}$$

Component

(Comp Component)

$$\frac{\Gamma \vdash \text{State}_j \;\therefore\; (x_j : T_j) \qquad (j = 1 .. n) \qquad \Gamma, x_1 : T_1 .. x_n : T_n \vdash \text{ExposedInterface}_k : \{I_k\} \qquad (k = 1 .. m)}{\Gamma \vdash \textit{COMPONENT}\ O\ \textit{IS}\ \text{State}_1 .. \text{State}_n\ \text{ExposedInterface}_1 .. \text{ExposedInterface}_m}$$

Declarations

(Decl Contains)

$$\frac{\Gamma_{CLSID} \vdash O : \{\} \qquad \Gamma \vdash I_j \in dom(\Gamma_{IID}) \quad (j = 1 .. n) \quad [x \notin dom(\Gamma)]}{\Gamma, \Gamma_{CLSID} \vdash \textit{CONTAINS}\ x : \{I_1 .. I_n\} = \textit{NEW}\ O \quad \therefore\ (x : \{I_1 .. I_n\})}$$

(Decl Method)

$$\frac{\Gamma, x_1 : T_1 .. x_n : T_n, \text{return} : T \vdash \text{body} \quad [x_1 .. x_n \notin dom(\Gamma)]}{\Gamma \vdash \textit{METHOD}\ M(x_1 : T_1 .. x_n : T_n) : T\ \textit{IS}\ \text{body} \ \therefore\ (M : T_1 .. T_n \rightarrow T)}$$

(Decl Const)

$$\frac{\Gamma \vdash E : T \quad [x \notin dom(\Gamma)]}{\Gamma \vdash (\text{CONST}\ x = E) \ \therefore\ (x : T)}$$

(Decl Var)

$$\frac{\Gamma \vdash T \quad [x \notin dom(\Gamma)]}{\Gamma \vdash \textit{VAR}\ x : T \ \therefore\ (x : T)}$$

Implementations

(Impl Aggregates)

$$\frac{\Gamma \vdash x : \{I\} \quad [\text{x is bound by Contains}]}{\Gamma \vdash \textit{AGGREGATES}\ x.I : \{I\}}$$

(Impl Interface)

$$\frac{\Gamma_{IID} \vdash I : \Gamma_I \quad [\Gamma_I = S_1 .. S_n] \qquad \Gamma_I \vdash \text{Method}_j \ \therefore\ S_j \quad (j = 1 .. n)}{\Gamma_{IID} \vdash \textit{INTERFACE}\ I\ \textit{IS}\ \text{Method}_1 .. \text{Method}_n : \{I\}}$$

Commands

(Cmd body)

$$\frac{\Gamma \vdash D_j \;\therefore\; S_j \quad (j = 1 \,..\, n) \qquad \Gamma,\, S_1 \,..\, S_n \vdash C}{\Gamma \vdash D_1 \,..\, D_n \;\; C}$$

(Cmd Sequence)

$$\frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash C_1 \,;\, C_2}$$

(Cmd Assign)

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash E : T \quad [\text{x is not bound by Contains}]}{\Gamma \vdash x := E}$$

(Cmd Return)

$$\frac{\Gamma \vdash E : T \qquad \Gamma \vdash \text{return} : T}{\Gamma \vdash \mathit{RETURN}\;\; E}$$

(Cmd If)

$$\frac{\Gamma \vdash E \;:\; \mathit{BOOLEAN} \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash \;\mathit{IF}\; E \;\; \mathit{THEN}\;\; C_1 \;\mathit{ELSE}\; C_2 \;\; \mathit{END}}$$

(Cmd While)

$$\frac{\Gamma \vdash E \;:\; \mathit{BOOLEAN} \quad \Gamma \vdash C}{\Gamma \vdash \mathit{WHILE}\;\; E \;\mathit{DO}\; C \;\mathit{END}}$$

(Cmd TypeCase)

$$\frac{\Gamma \vdash x : A_0 \qquad \Gamma,\, x : A \cup A_0 \vdash C_1 \qquad \Gamma \vdash C_2 \quad [\text{x is local variable}]}{\Gamma \vdash \;\mathit{CASE}\; x \;\mathit{IS}\; \{A\} \,:\, C_1 \;\mathit{ELSE}\; C_2 \;\; \mathit{END}}$$

Expressions

(Expr Plus)

$$\frac{\Gamma \vdash E_1 \;:\; \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 \;+\; E_2 :\; \mathit{INTEGER}}$$

(Expr Minus)

$$\frac{\Gamma \vdash E_1 \;:\; \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 \;-\; E_2 :\; \mathit{INTEGER}}$$

(Expr Or)

$$\frac{\Gamma \vdash E_1 \;:\; \mathit{BOOLEAN} \quad \Gamma \vdash E_2 : \mathit{BOOLEAN}}{\Gamma \vdash E_1 \;\mathit{OR}\; E_2 \;\;:\; \mathit{BOOLEAN}}$$

(Expr Equal)

$$\frac{\Gamma \vdash E_1 \;:\; T \quad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \;=\; E_2 \;\;:\; \mathit{BOOLEAN}}$$

(Expr NotEqual)

$$\frac{\Gamma \vdash E_1 \;:\; T \quad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \;\#\; E_2 \;\;:\; \mathit{BOOLEAN}}$$

(Expr Less)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 < E_2 : \mathit{BOOLEAN}}$$

(Expr Greater)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 > E_2 : \mathit{BOOLEAN}}$$

(Expr LessEqual)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 <= E_2 : \mathit{BOOLEAN}}$$

(Expr GreaterEqual)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 >= E_2 : \mathit{BOOLEAN}}$$

(Expr Multiply)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1 * E_2 : \mathit{INTEGER}}$$

(Expr Divide)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1\ \mathit{DIV}\ E_2 : \mathit{INTEGER}}$$

(Expr And)

$$\frac{\Gamma \vdash E_1 : \mathit{BOOLEAN} \quad \Gamma \vdash E_2 : \mathit{BOOLEAN}}{\Gamma \vdash E_1\ \mathit{AND}\ E_2 : \mathit{BOOLEAN}}$$

(Expr Mod)

$$\frac{\Gamma \vdash E_1 : \mathit{INTEGER} \quad \Gamma \vdash E_2 : \mathit{INTEGER}}{\Gamma \vdash E_1\ \mathit{MOD}\ E_2 : \mathit{INTEGER}}$$

(Expr New)

$$\frac{\Gamma_{CLSID} \vdash O : \{\}}{\Gamma \vdash \mathit{NEW}\ O : \{\}}$$

(Expr MethodCall)

$$\frac{\Gamma \vdash x : \{I\} \quad \Gamma_{IID} \vdash I : \Gamma_I \quad \Gamma_I \vdash M : T_1 \,..\, T_n \rightarrow T \qquad \Gamma \vdash E_j : T_j \ (j = 1 \,..\, n)}{\Gamma, \Gamma_{IID} \vdash x.I \rightarrow M(E_1 \,..\, E_n) : T}$$

(Expr Number)

$$\frac{\Gamma \vdash \lozenge}{\Gamma \vdash \mathrm{Int} : \mathit{INTEGER}}$$

(Expr Null)

$$\frac{\Gamma \vdash \lozenge}{\Gamma \vdash \mathit{NIL} : \{\}}$$

(Expr False)

$$\frac{\Gamma \vdash \lozenge}{\Gamma \vdash \mathit{FALSE} : \mathit{BOOLEAN}}$$

(Expr True)

$$\frac{\Gamma \vdash \lozenge}{\Gamma \vdash \mathit{TRUE} : \mathit{BOOLEAN}}$$

6. Conclusion

In this paper we introduced an approach to the formalization of COM, based on a model language named COMEL. We introduced a formal type system for COMEL. Using conventional programming languages, most of COM's rules need to be supported in each object. In COM, it is required that every interface keeps a reference count on it. This places a burden on a client of a COM interface to call the AddRef function for every new copy of an interface pointer and to call the Release function for every destruction of an interface pointer (except where special rules explicitly demand otherwise). COM's reference-counting scheme is the most error-prone aspect of COM programming. To be safe, garbage collection needs to be done automatically. In COMEL, garbage collection is part of the language's operational semantics (not presented in this paper). This approach is also taken in some real-world programming languages with COM bindings, such as Microsoft's Visual J++ [MVJ97] and Oberon Microsystems's Direct-To-COM (DTC) compiler [DTC96].

COM also places a burden on clients wishing to create new COM objects. It is a responsibility of a client using COM to create a new object by passing the CLSID as a parameter to the CoCreateInstance function and to initialize the object explicitly before using the object's services. Object creation in COMEL is captured declaratively by the *NEW* operator. In COM, creation and initialization of a component object are kept separate. This is fundamental to COM and therefore is kept in COMEL.

Most of COM's rules are captured at the declaration level of COMEL. This is also the case for containment and aggregation. In particular, the complex COM rules for setting up an aggregate are captured in the operational semantics of COMEL's *AGGREGATES* construct.

Our work is still progressing on the operational semantics of COMEL and the soundness proof for the COMEL type system.

Acknowledgments

We would like to thanks Chris Ho-Stuart and Paul Roe for their useful comments and reviews on earlier drafts of the paper.

References

[Abad96] Abadi M. and Cardelli L. *A Theory of Objects.* Springer, 1996.

[Betz94] Betz M. *Interoperable Objects.* Dr. Dobb's Journal, Vol. 19, No. 11, October 1994.

[Broc95] Brockschmidt K. *Inside OLE*. Microsoft Press, second edition, 1995.

[Card85] Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Vol. 17, No. 4, December 1985.

[Card89] Cardelli, L. *Typeful Programming.* Tech. Rep. 45, DEC System Research Center, May 1989.

[Card96] Cardelli L. *Type Systems.* In The Handbook of Computer Science and Engineering, Allen B. Tucker Jr. Ed., Chapter 103, 2208-2236, CRC Press, 1997.

[Chap96] Chappell D. *Understanding ActiveX™ and OLE.* Microsoft Press, 1996.

[DTC96] Direct-To-COM compiler. *http://www.oberon.ch/prod/dtc_docu.html.* Oberon microsystems, Inc., Switzerland 1996.

[Micr95] Microsoft. *The COM Specification.* Draft, Microsoft Corporation, 1996.

[MVJ97] Microsoft's Visual J++. *http://www.microsoft.com/visualj/.* Microsoft Corporation, 1997.

[Pals94] Palsberg, J. and Schwartzbach, M.I. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[Plot81] Plotkin G.D. *A Structural Approach to Operational Semantics.* DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981 (reprinted 1991).

[Roge97] Rogerson D. *Inside COM.* Microsoft Press, 1997.

[Schm94] Schmidt D.A. *The Structure of Typed Programming Languages.* The MIT Press, 1994.

[Szyp96] Szyperski, C. *Independently Extensible Systems - Software Engineering Potential and Challenges.* In Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 1996.

[Wins94] Winskel G. *The Formal Semantics of Programming Languages.* The MIT Press, second edition, 1994.